Basics of Deductive Program Verification

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

December 7th, 2021

Preliminaries

Very first question
Lectures in English or in French?

▶ Schedule on the Web page https://marche.gitlabpages.inria.fr/lecture-deductive-verif/
▶ Lectures 1,2,3,4: Claude Marché
▶ Lectures 5,6,7,8: Jean-Marie Madiot
▶ Evaluation:
  ▶ project $P$ using the Why3 tool (http://why3.lri.fr)
  ▶ final exam $E$: date to decide
  ▶ final mark = $(2E + P + \max(E,P))/4$
▶ Project:
  ▶ provided at the beginning of January
  ▶ No lecture on February 1st, replaced by practical lab session (support for project)
  ▶ due date around mid-February
▶ Internships (stages)

Outline

Introduction, Short History
Preliminary on Automated Deduction
  Classical Propositional Logic
  First-order logic
  Logic Theories
  Limitations of Automatic Provers
Introduction to Deductive Verification
  Formal contracts
  Hoare Logic
  Dijkstra’s Weakest Preconditions
Exercises

General Objectives

Ultimate Goal

Verify that software is free of bugs

Famous software failures:
http://www.cs.tau.ac.il/~nachumd/horror.html

This lecture

Computer-assisted approaches for verifying that a software conforms to a specification
Some general approaches to Verification

Static analysis, Algorithmic Verification
- **model checking** (automata-based models)
- **abstract interpretation** (domain-specific model, e.g. numerical)

Deductive verification
- formal models using expressive logics
- verification = computer-assisted mathematical proof

Refinement
- Formal models
- Code derived from model, correct by construction

A long time before success

Computer-assisted verification is an old idea
- Turing, 1948
- Floyd-Hoare logic, 1969

Success in practice: only from the mid-1990s
- Importance of the *increase of performance of computers*

A first success story:
- Paris metro line 14, using Atelier B (1998, refinement approach)

Other Famous Success Stories

  http://www.astree.ens.fr/
- Microsoft's hypervisor: using Microsoft's VCC and the Z3 automated prover (2008, deductive verification)
- More recently: verification of PikeOS
- Certified C compiler, developed using the Coq proof assistant (2009, correct-by-construction code generated by a proof assistant)
  http://compcert.inria.fr/
- L4.verified micro-kernel, using tools on top of Isabelle/HOL proof assistant (2010, Haskell prototype, C code, proof assistant)
Other Success Stories at Industry

- Frama-C
- EDF: abstract interpretation
- Airbus: deductive verification
- Spark/Ada: Verification of Ada programs

https://www.adacore.com/industries

Remark
The two above use Why3 internally

Outline

Introduction, Short History

Preliminary on Automated Deduction
- Classical Propositional Logic
- First-order logic
- Logic Theories
- Limitations of Automatic Provers

Introduction to Deductive Verification
- Formal contracts
- Hoare Logic
- Dijkstra’s Weakest Preconditions

Exercises

Proposition logic in a nutshell

- Syntax:

\[ \phi ::= \bot \mid \top \mid A, B \quad (\text{atomic}) \]
\[ \mid \phi \land \phi \mid \phi \lor \phi \mid \lnot \phi \]
\[ \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi \]

- Semantics, models: truth tables

- \( \phi \) is satisfiable if it has a model
- \( \phi \) is valid if true in all models
  (equivalently \( \lnot \phi \) is not satisfiable)

SAT is decidable \( \iff \) SAT solvers

Demo with Why3

\$ why3 ide propositional.mlw

Notice that Why3 indeed queries solvers for satisfiability of \( \lnot \phi \)

Focus on the “Tools” menu of Why3
First-order logic in a nutshell

▶ Syntax:

\[ \varphi ::= \cdots \mid P(t, \ldots, t) \quad (\text{predicates}) \mid \forall x. \phi \mid \exists x. \phi \]

\[ t ::= x \quad \text{variables} \mid f(t, \ldots, t) \quad (\text{function symbols}) \]

▶ Semantics: models must interpret variables. C

▶ Satisfiability undecidable, but still semi-decidable: there exists complete systems of deduction rules (sequent calculus, natural deduction, superposition calculus)

▶ Examples of solvers: E, Spass, Vampire

Implement refutationally complete procedure: if they answer 'unsat' then formula is unsatisfiable

Demo with Why3

first-order.mlw

Notice that Why3 logic is typed, and application is curried

Logic Theories

▶ Theory = set of formulas (called theorems) closed by logical consequence

▶ Axiomatic Theory = set of formulas generated by axioms (or axiom schemas)

▶ Consistent Theory

for no \( P, P \) and \( \neg P \) are both theorems

equivalently: ‘false’ is not a theorem

equivalently: the theory has models

▶ Consistent Axiomatization

‘false’ is not derivable

Theory of Equality

\[ \forall x. x = x \]

\[ \forall x, y. x = y \rightarrow y = x \]

\[ \forall x, y, z. x = y \land y = z \rightarrow x = z \]

(congruence) for all function symbols \( f \) of arity \( k \):

\[ \forall x_1, y_1, \ldots, x_k, y_k. x_1 = y_1 \land \cdots \land x_k = y_k \rightarrow f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k) \]

and for all predicates \( p \) of arity \( k \):

\[ \forall x_1, y_1, \ldots, x_k, y_k. x_1 = y_1 \land \cdots \land x_k = y_k \rightarrow p(x_1, \ldots, x_k) \rightarrow p(y_1, \ldots, y_k) \]

Theory of Equality, Continued

\[ \forall x. x = x \]

\[ \forall x, y. x = y \rightarrow y = x \]

\[ \forall x, y, z. x = y \land y = z \rightarrow x = z \]

(congruence) \( \cdots \)

▶ General first-order deduction bad in such a case \( \Rightarrow \) dedicated methods

▶ paramodulation

▶ congruence closure (for quantifier-free fragment)

▶ SMT solvers (Alt-Ergo, CVC4, Z3) implement dedicated (semi-)decision procedures

Demo with Why3
equality.mlw
Theories Continued

**Theory of a given model**
= formulas true in this model

- Central example: theory of linear integer arithmetic, i.e. formulas using $+$ and $\leq$
  - First-order theory is known to be decidable (Presburger)
  - SMT solvers typically implement a procedure for the existential fragment
- Also: theory of (non-linear) real arithmetic is decidable (Tarski)

Non-linear Integer Arithmetic

(a.k.a. Peano Arithmetic)

**First-Order Integer Arithmetic**
All valid first-order formulas on integers with $+$, $\times$ and $\leq$

- This theory is not even semi-decidable
- SMT solvers implement incomplete satisfiability checks: if solver answers 'unsat' then it is unsatisfiable

Demo with Why3
arith.mlw

Digression about Non-linear Integer Arithmetic

**Representation Theorem (Gödel)**
Every recursive function $f$ is representable by a predicate $\phi_f$ such that

$$\phi_f(x_1, \ldots, x_k, y)$$

is true if and only if

$$y = f(x_1, \ldots, x_k)$$

**First incompleteness Theorem (Gödel)**
That theory is not recursively axiomatizable

Summary of prover limitations

- Superposition solvers (E, Spass, )
  - do not support well theories except equality
  - do quite well with quantifiers
- SMT solvers (Alt-Ergo, CVC4, Z3)
  - several theories are built-in
  - weaker with quantifiers
- None support reasoning by induction
Outline

Introduction, Short History
Preliminary on Automated Deduction
  Classical Propositional Logic
  First-order logic
  Logic Theories
  Limitations of Automatic Provers
Introduction to Deductive Verification
  Formal contracts
  Hoare Logic
  Dijkstra’s Weakest Preconditions
Exercises

IMP language

IMP language
A very basic imperative programming language
  only global variables
  only integer values for expressions
  basic statements:
    - assignment $x \leftarrow e$
    - sequence $S_1; S_2$
    - conditionals if $e$ then $S_1$ else $S_2$
    - loops while $e$ do $S$
    - no-op skip

Formal Contracts

General form of a program:

Contract
  - precondition: expresses what is assumed before running the program
  - post-condition: expresses what is supposed to hold when program exits

Demo with Why3
contracts.mlw

Hoare triples

$Hoare triple$ : notation $\{P\} s \{Q\}$
- $P$ : formula called the precondition
- $Q$ : formula called the postcondition

Intended meaning

$\{P\} s \{Q\}$ is true if and only if:
when the program $s$ is executed in any state satisfying $P$, then
(if execution terminates) its resulting state satisfies $Q$

This is a $Partial Correctness$: we say nothing if $s$ does not terminate
Examples

Examples of valid triples for partial correctness:
- \{ x = 1 \} x < - x + 2 \{ x = 3 \}
- \{ x = y \} x < - x + y \{ x = 2 + y \}
- \{ \exists v. x = 4 * v \} x < - x + 42 \{ \exists w. x = 2 * w \}
- \{ true \} while 1 do skip \{ false \}

Running Example

Three global variables \( n, \text{count}, \text{and sum} \)

```plaintext
count <- 0; sum <- 1;
while sum <= n do
  count <- count + 1; sum <- sum + 2 * count + 1
```

What does this program compute?
(assuming input is \( n \) and output is \( \text{count} \))

Informal specification:
- at the end of execution of this program, \( \text{count} \) contains the square root of \( n \), rounded downward
- e.g. for \( n = 42 \), the final value of \( \text{count} \) is 6.

See file `imp_isqrt.mlw`

Hoare logic as an Axiomatic Semantics

Original Hoare logic [\(~ 1970\)]

Axiomatic Semantics of programs

Set of inference rules producing triples

\[
\begin{align*}
\{ P \} \text{skip} \{ P \} \\
\{ P \} x \leftarrow e \{ x \rightarrow e \{ P \} \} \\
\{ P \} s_1 \{ Q \} \quad \{ Q \} s_2 \{ R \} \\
\{ P \} s_1 ; s_2 \{ R \}
\end{align*}
\]

Notation \( P[x \leftarrow e] \): replace all occurrences of program variable \( x \) by \( e \) in \( P \).

Hoare Logic, continued

Frame rule:

\[
\frac{\{ P \} s \{ Q \}}{\{ P \land R \} s \{ Q \land R \}}
\]

with \( R \) a formula where no variables assigned in \( s \) occur

Consequence rule:

\[
\frac{\{ P' \} s \{ Q' \} \quad \models P \rightarrow P'}{\{ P \} s \{ Q \} \quad \models Q' \rightarrow Q}
\]

Example: proof of

\[
\{ x = 1 \} x \leftarrow x + 2 \{ x = 3 \}
\]
Proof of the example

\[
\begin{align*}
\frac{x + 2 = 3 \quad \models x = 1 \rightarrow x + 2 = 3}{\frac{x = 3 \quad \models x = 3}{\models \{x = 1 \} \lhd x + 2 \{x = 3\}}}
\end{align*}
\]

Informal justification of the while rule

\[
\begin{align*}
\frac{\{I \land e\} \, s\{I\}}{\models \{I\} \text{while } e \, s\{I \land \neg e\}}
\end{align*}
\]

Rules for if and while:

\[
\begin{align*}
\{P \land e\} \, s_1 \{Q\} & \quad \{P \land \neg e\} \, s_2 \{Q\} \\
\{P\} \text{if } e \text{ then } s_1 \text{ else } s_2 \{Q\} & \\
\{I \land e\} \, s\{I\} & \\
\{I\} \, \text{while } e \, s\{I \land \neg e\}
\end{align*}
\]

Example: isqrt(42)

Exercise: prove of the triple
\[
\{n \geq 0\} \text{ISQRT} \{\text{count}^2 \leq n \land n < (\text{count} + 1)^2\}
\]

Could we do that by hand?

Back to demo: file imp_isqrt.mlw

Warning

Finding an adequate loop invariant is a major difficulty
Operational semantics

PLOTIN 1981, STRUCTURAL OPERATIONAL SEMANTICS (SOS)

- we use a standard small-step semantics

- program state: describes content of global variables at a given time. It is a finite map \( \Sigma \) associating to each variable \( x \) its current value denoted \( \Sigma(x) \).

- Value of an expression \( e \) in some state \( \Sigma \):
  - denoted \( [e]_{\Sigma} \)
  - always defined, by the following recursive equations:

\[

\begin{align*}
[n]_{\Sigma} &= n \\
[x]_{\Sigma} &= \Sigma(x) \\
[e_1 \text{ op } e_2]_{\Sigma} &= [e_1]_{\Sigma} \text{ op } [e_2]_{\Sigma}
\end{align*}
\]

- \([\text{op}]\) natural semantic of operator \(\text{op}\) on integers (with relational operators returning \(0\) for false and \(\neq 0\) for true).

Semantics of statements

Semantics of statements: defined by judgment

\[ \Sigma, s \xrightarrow{} \Sigma', s' \]

meaning: in state \( \Sigma \), executing one step of statement \( s \) leads to the state \( \Sigma' \) and the remaining statement to execute is \( s' \).

The semantics is defined by the following rules.

\[
\begin{align*}
\Sigma, x = e \xrightarrow{} \Sigma\{ x \leftarrow [e]_{\Sigma}\}, \text{skip} \\
\Sigma, s_1 \xrightarrow{} \Sigma', s'_1 \\
\Sigma, (s_1; s_2) \xrightarrow{} \Sigma', (s'_1; s_2) \\
\Sigma, (\text{skip}; s) \xrightarrow{} \Sigma, s
\end{align*}
\]
Execution of programs

- $\rightsquigarrow$: a binary relation over pairs (state, statement)
- Transitive closure: $\rightsquigarrow^+$
- Reflexive-transitive closure: $\rightsquigarrow^*$

In other words:

$$\Sigma, s \rightsquigarrow^* \Sigma', s'$$

means that statement $s$, in state $\Sigma$, reaches state $\Sigma'$ with remaining statement $s'$ after executing some finite number of steps.

Running example:

$$\{ n = 42, count = ?, sum = ? \}, ISQR\rightarrow \rightsquigarrow^*$$

$$\{ n = 42, count = 6, sum = 49 \}, skip$$

Execution and termination

- any statement except skip can execute in any state
- the statement skip alone represents the final step of execution of a program
- there is no possible runtime error.

Definition

Execution of statement $s$ in state $\Sigma$ terminates if there is a state $\Sigma'$ such that $\Sigma, s \rightsquigarrow^* \Sigma', skip$.

- since there are no possible runtime errors, $s$ does not terminate means that $s$ diverges (i.e., executes infinitely).

Semantics of formulas

- $\llbracket p \rrbracket_\Sigma$: semantics of formula $p$ in program state $\Sigma$
- is a logic formula where no program variables appear anymore
- defined recursively as follows.

  $$\llbracket e \rrbracket_\Sigma = \begin{cases} \top & \text{if } \llbracket e \rrbracket_\Sigma \neq 0 \\ \bot & \text{otherwise} \end{cases}$$
  $$\llbracket p_1 \land p_2 \rrbracket_\Sigma = \llbracket p_1 \rrbracket_\Sigma \land \llbracket p_2 \rrbracket_\Sigma$$
  $$\vdots$$

where semantics of expressions is augmented with

$$\llbracket \nu \rrbracket_\Sigma = \nu$$
$$\llbracket x \rrbracket_\Sigma = \Sigma(x)$$

Notations:

- $\Sigma \models p$: the formula $\llbracket p \rrbracket_\Sigma$ is valid
- $\models p$: formula $\llbracket p \rrbracket_\Sigma$ holds in all states $\Sigma$. 

Semantics of formulas

Other presentation of the semantics: $\llbracket p \rrbracket_\Sigma$:

- inline semantic of first-order formula
- $\llbracket e \rrbracket_{\Sigma, \nu}$ with $\nu$ mapping of logic variables to integers.

- defined recursively as follows.

  $$\llbracket p_1 \land p_2 \rrbracket_{\Sigma, \nu} = \begin{cases} \top & \text{if } \llbracket p_1 \rrbracket_{\Sigma, \nu} = \top \text{ and } \llbracket p_2 \rrbracket_{\Sigma, \nu} = \top \\ \bot & \text{otherwise} \end{cases}$$
  $$\llbracket \forall x. e \rrbracket_{\Sigma, \nu} = \begin{cases} \top & \text{if for all } \nu. \llbracket e \rrbracket_{\Sigma, \nu[x \leftarrow \nu]} = \top \\ \bot & \text{otherwise} \end{cases}$$
  $$\vdots$$

where semantics of expressions is augmented with

$$\llbracket \nu \rrbracket_{\Sigma, \nu} = \nu(\nu)$$
$$\llbracket x \rrbracket_{\Sigma, \nu} = \Sigma(x)$$
Soundness

**Definition (Partial correctness)**

Hoare triple \( \{ P \} s \{ Q \} \) is said **valid** if:

for any states \( \Sigma, \Sigma' \), if

- \( \Sigma, s \Rightarrow^* \Sigma' \), skip and
- \( \Sigma \models P \)

then \( \Sigma' \models Q \)

**Theorem (Soundness of Hoare logic)**

The set of rules is correct: any derivable triple is valid.

This is **proved by induction on the derivation tree** of the considered triple.

For each rule: assuming that the triples in premises are valid, we show that the triple in conclusion is valid too.

Digression: Completeness of Hoare Logic

Two major difficulties for proving a program

- **guess the appropriate intermediate formulas** (for sequence, for the loop invariant)
- **prove the logical premises of consequence rule**

Theoretical question: completeness. Are all valid triples derivable from the rules?

**Theorem (Relative Completeness of Hoare logic)**

The set of rules of Hoare logic is relatively complete: if the logic language is expressive enough, then any valid triple \( \{ P \} s \{ Q \} \) can be derived using the rules.


Yet, this does not provide an effective recipe to discover suitable loop invariants (see also the theory of abstract interpretation [Cousot, 1990])

Annotated Programs

**Goal**

Add automation to the Hoare logic approach

We augment IMP with **explicit loop invariants**

\[ \text{while } \theta \text{ invariant } \land \text{ do } s \]

Weakest liberal precondition

[Dijkstra 1975]

**Function** \( \text{WLP}(s, Q) \):

- \( s \) is a statement
- \( Q \) is a formula
- returns a formula

It should return the **minimal precondition** \( P \) that validates the triple \( \{ P \} s \{ Q \} \)
**Definition of WLP(s, Q)**

Recursive definition:

\[
\begin{align*}
\text{WLP}(&\text{skip}, Q) = Q \\
\text{WLP}(x < e, Q) = Q[x \leftarrow e] \\
\text{WLP}(s_1; s_2, Q) = \text{WLP}(s_1, \text{WLP}(s_2, Q)) \\
\text{WLP}(\text{if } e \text{ then } s_1 \text{ else } s_2, Q) = \\
(e \rightarrow \text{WLP}(s_1, Q)) \land (\neg e \rightarrow \text{WLP}(s_2, Q))
\end{align*}
\]

Examples

\[
\text{WLP}(x < x + y, x = 2y) \equiv x + y = 2y
\]

\[
\text{WLP(while } y > 0 \text{ invariant even}(y) \text{ do } y < y - 2, \text{even}(y)) \equiv
\text{even}(y) \land \\
\forall v, ((v > 0 \land \text{even}(v)) \rightarrow \text{even}(v - 2)) \land ((v \leq 0 \land \text{even}(v)) \rightarrow \text{even}(v))
\]

**Definition of WLP(s, Q), continued**

\[
\begin{align*}
\text{WLP}(\text{while } e \text{ invariant } I \text{ do } s, Q) = \\
I \land \forall v_1, \ldots, v_k. \\
((e \land I) \rightarrow \text{WLP}(s, I)) \land ((\neg e \land I) \rightarrow Q)[w_i \leftarrow v_i]
\end{align*}
\]

where \(w_1, \ldots, w_k\) is the set of assigned variables in statement \(s\) and \(v_1, \ldots, v_k\) are fresh logic variables

**Soundness**

**Theorem (Soundness)**

For all statement \(s\) and formula \(Q\), \(\{\text{WLP}(s, Q)\} s\{Q\}\) is valid.

Proof by induction on the structure of statement \(s\).

**Consequence**

For proving that a triple \(\{P\} s\{Q\}\) is valid, it suffices to prove the formula \(P \rightarrow \text{WLP}(s, Q)\).

This is basically the goal that Why3 produces.
Exercise 1

Consider the following (inefficient) program for computing the sum $a + b$.

```plaintext
x <- a; y <- b;
while y > 0 do
  x <- x + 1; y <- y - 1
```

(Why3 file to fill in: imp_sum.mlw)

- Propose a post-condition stating that the final value of $x$ is the sum of the values of $a$ and $b$
- Find an appropriate loop invariant
- Prove the program.

Exercise 2

The following program is one of the original examples of Floyd.

```plaintext
q <- 0; r <- x;
while r >= y do
  r <- r - y; q <- q + 1
```

(Why3 file to fill in: imp_euclide.mlw)

- Propose a formal precondition to express that $x$ is assumed non-negative, $y$ is assumed positive, and a formal post-condition expressing that $q$ and $r$ are respectively the quotient and the remainder of the Euclidean division of $x$ by $y$.
- Find appropriate loop invariants and prove the correctness of the program.

Exercise 3

Let's assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of $x$ by 2 and its remainder. The following program is supposed to compute, in variable $r$, the power $x^n$.

```plaintext
r <= 1; p <- x; e <- n;
while e > 0 do
  if mod2(e) <> 0 then r <- r * p;
  p <- p * p;
  e <- div2(e);
```

(Why3 file to fill in: power_int.mlw)

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.
Exercise 4

The Fibonacci sequence is defined recursively by \( \text{fib}(0) = 0, \text{fib}(1) = 1 \) and \( \text{fib}(n + 2) = \text{fib}(n + 1) + \text{fib}(n) \). The following program is supposed to compute \( \text{fib} \) in linear time, the result being stored in \( y \).

\[
y \leftarrow 0; \quad x \leftarrow 1; \quad i \leftarrow 0;
\text{while } i < n \text{ do}
\quad \text{aux} \leftarrow y; \quad y \leftarrow x; \quad x \leftarrow x + \text{aux}; \quad i \leftarrow i + 1
\]

▶ Assuming \( \text{fib} \) exists in the logic, specify appropriate pre- and post-conditions.
▶ Prove the program.

Exercise (original Floyd rule for assignment)

1. Prove that the triple

\[
\{ P \} x \leftarrow e \{ \exists v, e[x \leftarrow v] = x \land P[x \leftarrow v] \}
\]

is valid with respect to the operational semantics.

2. Show that the triple above can be proved using the rules of Hoare logic.

Let us assume that we replace the standard Hoare rule for assignment by the Floyd rule

\[
\{ P \} x \leftarrow e \{ \exists v, e[x \leftarrow v] = x \land P[x \leftarrow v] \}
\]

3. Show that the triple \( \{ P[x \leftarrow e] \} x \leftarrow e(P) \) can be proved with the new set of rules.

Bibliography


